VanillaCNN:
Deriving Convolutional Neural Networks from Scratch

Ngan Vu
Final Project, S&DS 430: Optimization Techniques
Yale University

December 13, 2018

Contents

1 Introduction

1.1 What is a Convolutional Neural Network?

Convolutional Neural Network (CNNs) are a class of **Deep Neural Networks** (DNNs) and are commonly used in image analysis tasks. Like all DNNs, CNNs aim to mimic the information processing and communication patterns in the **biological nervous system** - how neurons interact with one another to accomplish a task. CNNs, in particular, are inspired by the connectivity patterns of neurons specifically in the **animal visual cortex**, in which individual cortical neurons respond to stimuli only in a restricted region of the visual field known as the **receptive field**.

A CNN typically consists of an **input layer**, an **output layer**, and multiple **hidden layers**. The first hidden layers are typically pairs of **convolutional layers** and **pooling layers**. The last hidden layers are typically **fully connected layers**. On a high level:

- **Convolutional layers** convolve one or more inputs (2D grids of cells) in the previous layer with one or more kernels, resulting in one or more outputs (2D grids of cells) in the next layer. The convolution operation is defined below.
- **Pooling layers** combine multiple cells from the input in the previous layer into one cell in the output in the next layer. The combination operation is typically summing, averaging, or taking the max.
- **Fully connected layers** connect every cell in the previous layer with every cell in the next layer. Each connection is assigned a weight, which specifies how much each input cell contributes to each output cell.

In each **learning iteration** (or **epoch**), an input is passed through the network in a process called **feed forward** to get a predicted output. The predicted output is then compared against the actual output to determine a **loss**. Then, the **gradient** of the loss is propagated back through the network in a process called **back propagation** to evaluate the partial derivatives of the parameters being learned (kernels, weights, and biases). Each parameter is then adjusted in the direction of negative partial derivative to reduce the loss. This method of following the negative gradient direction to minimize loss is called **gradient descent**. The whole process is repeated for many iterations until the loss converges, which suggests that a local minimum has been achieved.

1.2 Motivation of this paper

Currently, while there are a lot of materials on Convoluational Neural Networks available online, to my knowledge, none of them concretely derives how CNNs work and then implements a CNN using their derivation. Articles that go through the math of CNNs do not provide any implementation. This lack of accompanying code allows the notations to be somewhat lax, especially in indexing (for example, allowing negative indices in a matrix, or not starting the indices from the top-left corner of a matrix). Articles that actually implement their derivations only use toy data sets whose data points are very small (2×2 or 3×3 images).

In practice, CNNs are usually built with Python packages and all operations (feed forward, loss evaluation, backpropagation, parameters update) are handled by these packages. Users of these packages do not necessarily understand how CNNs work behind the scene.

This paper aims to **concretely derives the math** behind Convolutional Neural Networks and then **implements an instance** of CNN using this derivation. With such a goal comes many stricter requirements, including but not limited to:

- enforcing the consistency of the sizes of all inputs, outputs, kernels, weights, and biases
- starting all indices with 0 and allowing no negative indices

so that **all mathematical formulas can be translated directly to code**. I believe this step-by-step derivation, accompanied by executable code, is helpful for both beginners and experienced readers who want to understand CNN more thoroughly.

2 Mathematical Background

2.1 Notations

- Vector: $\mathbf{v}$
- Matrix: $\boldsymbol{A}$
- Row i of matrix: $\boldsymbol{A}(\boldsymbol{i}, \boldsymbol{:})$; Column j of matrix: $\boldsymbol{A}(\boldsymbol{:}, \boldsymbol{j})$
- Convolution: $\boldsymbol{I} * \boldsymbol{K}$
- Correlation: $\boldsymbol{I} \star \boldsymbol{K}$
- Sigmoid function: $\sigma(x)$
- Pointwise product of matrices: $\boldsymbol{A} \odot \boldsymbol{B}$
- Dot product: $\bullet$
- Multiplication of scalars/matrices: ab or $\boldsymbol{AB}$, although sometimes $a \cdot b$ or $\boldsymbol{A} \cdot \boldsymbol{B}$ is used.
- Indexing: All indices start with 0 to follow the convention in most programming languages.
- Set of integers from 0 to $N - 1$: $[N]$

2.2 Correlation and Convolution

In two dimensions, both **correlation** (also called cross-correlation) and **convolution** are processes of applying a 2D **kernel** on each cell of a 2D **input**, resulting in a 2D **output** that expresses how the input is modified by the kernel. The main difference between convolution and correlation is the **orientation of the kernel**.

We will discuss both operations below to see why it is actually better to replace the convolution operation in each convolutional layer of a CNN with the correlation operation, while not changing the functionality of the CNN.

2.2.1 Definition

With a $(x_i \times y_i)$ input $\boldsymbol{I}$, a $(x_k \times y_k)$ kernel $\boldsymbol{K}$, and a $(x_o \times y_o)$ output $\boldsymbol{O}$:

- The formula for the **correlation** is:

$$\boldsymbol{O}(i,j) = \boldsymbol{I} \star \boldsymbol{K} = \sum_{u=0}^{x_k-1} \sum_{v=0}^{y_k-1} \boldsymbol{I}(i+u, j+v)\boldsymbol{K}(u,v) \quad \forall i \in [x_o], j \in [y_o] \tag{1}$$

- The formula for the **convolution** is:

$$\boldsymbol{O}(i,j) = \boldsymbol{I} * \boldsymbol{K} = \sum_{u=0}^{x_k-1} \sum_{v=0}^{y_k-1} \boldsymbol{I}(i-u, j-v)\boldsymbol{K}(u,v) \quad \forall i \in [x_o], j \in [y_o] \tag{2}$$

Correlation of $\boldsymbol{I}$ and $\boldsymbol{K}$ is actually equivalent to convolution of $\boldsymbol{I}$ and $\boldsymbol{K_{rot}}$, where $\boldsymbol{K_{rot}}$ is $\boldsymbol{K}$ rotated by 180^o. This is because:

$$\begin{aligned}
\boldsymbol{K}_{rot}(u,v) &= \boldsymbol{K}(x_k - 1 - u, y_k - 1 - v) \\
\boldsymbol{O}_{conv}(i,j) &= \sum_{u=0}^{x_k-1} \sum_{v=0}^{y_k-1} \boldsymbol{I}(i-u, j-v)\boldsymbol{K}(u,v) \\
&= \sum_{(x_k-1-u)=0}^{x_k-1} \sum_{(y_k-1-v)=0}^{y_k-1} \boldsymbol{I}(i-(x_k-1-u), j-(y_k-1-v))\boldsymbol{K}(x_k-1-u, y_k-1-v) \\
&\quad \text{(replace } u \text{ with } (x_k-1-u) \text{ and } v \text{ with } (y_k-1-v)) \\
&= \sum_{u=0}^{x_k-1} \sum_{v=0}^{y_k-1} \boldsymbol{I}((i-x_k-1)+u, (j-y_k-1)+v)\boldsymbol{K}_{rot}(u,v) \\
&\quad \text{(rewrite the bounds with the exactly equivalent bounds)} \\
&= \boldsymbol{O}_{corr}((i-x_k-1), (j-y_k-1))
\end{aligned}$$

As we could see, there is a bijective mapping between $\boldsymbol{O}_{conv}$ and $\boldsymbol{O}_{corr}$. The difference in indices is due to the complication in indexing in convolution using the formula above; this complication is addressed below. The final results of $\boldsymbol{I} * \boldsymbol{K}$ and $\boldsymbol{I} \star \boldsymbol{K_{rot}}$ should be exactly the same.

2.2.2 Why replace convolution with correlation?

The main purpose of this section is to explain why I use correlation instead of convolution in my derivation and implementation of CNNs. We saw above that convolution is just correlation with the kernel rotated 180^o. This section shows that correlation is easier to implement than convolution and therefore if we can replace convolution with correlation, we should do so.

Let's look at the indices and the shapes of the input $\boldsymbol{I}$, output $\boldsymbol{O}$, and kernel $\boldsymbol{K}$.

- Correlation:
  According to equation (1), to make sure that the indices of $\boldsymbol{I}$ is not out of bound and that all indices in $\boldsymbol{I}$ is touched, we require that:
  - $\forall i \in [x_o], u \in [x_k] : 0 \leq i + u \leq x_i - 1$
    $\Rightarrow \begin{cases} 0 = \min(i) + \min(u) = 0 + 0, \text{ already satisfied} \\ x_i - 1 = \max(i) + \max(u) = (x_o - 1) + (x_k - 1) = x_o + x_k - 2 \end{cases}$
    $\Rightarrow x_o = x_i - x_k + 1$
  - $\forall j \in [y_o], v \in [y_k] : 0 \leq j + v \leq y_j - 1$
    $\Rightarrow \begin{cases} 0 = \min(i) + \min(u) = 0 + 0, \text{ already satisfied} \\ y_i - 1 = \max(j) + \max(v) = (y_o - 1) + (y_k - 1) = y_o + y_k - 2 \end{cases}$
    $\Rightarrow y_o = y_i - y_k + 1$
- Convolution:
  Note that because there is subtraction, for the convolution formula to work in our implementation, we need to make either of these two sacrifices in indexing:
  - Indices of $\boldsymbol{I}$ are non-negative, but indices of $\boldsymbol{O}$ do not start at 0 (they start at $x_k - 1$ and $y_k - 1$). In this case:
    * For $\boldsymbol{I}$: $\begin{cases} 0 \leq i - u \leq x_i - 1 \\ 0 \leq j - v \leq y_i - 1 \end{cases}$ (normal indices)
    * For $\boldsymbol{O}$: $\begin{cases} x_k - 1 \leq i \leq x_o + x_k - 2 \\ y_k - 1 \leq j \leq y_o + y_k - 2 \end{cases}$ (shift indices up by $(x_k - 1, y_k - 1)$)
  - Indices of $\boldsymbol{O}$ do start at 0, but indices of $\boldsymbol{I}$ are not non-negative (they start at $-x_k + 1$ and $-y_k + 1$). In this case:
    * For $\boldsymbol{I}$: $\begin{cases} -x_k + 1 \leq i - u \leq x_i - x_k \\ -y_k + 1 \leq j - v \leq y_i - y_k \end{cases}$ (shift indices down by $(x_k - 1, y_k - 1)$)
    * For $\boldsymbol{O}$: $\begin{cases} 0 \leq i \leq x_o - 1 \\ 0 \leq j \leq y_o - 1 \end{cases}$ (normal indices)

  We will use the second option so that we have $i \in [x_o], j \in [y_o]$.
  According to equation (2), to make sure that the indices of $\boldsymbol{I}$ is not out of bound and that all indices in $\boldsymbol{I}$ is touched, we require that:
  - $\forall i \in [x_o], u \in [x_k] : -x_k + 1 \leq i + u \leq x_i - x_k$
    $\Rightarrow \begin{cases} -x_k + 1 = \min(i) - \max(u) = 0 - (x_k + 1), \text{ already satisfied} \\ x_i - x_k = \max(i) - \min(u) = (x_o - 1) - 0 = x_o - 1 \end{cases}$
    $\Rightarrow x_o = x_i - x_k + 1$
  - $\forall j \in [y_o], v \in [y_k] : -y_k + 1 \leq j + v \leq y_i - y_k$
    $\Rightarrow \begin{cases} -y_k + 1 = \min(i) + \min(u) = 0 - (y_k - 1), \text{ already satisfied} \\ y_i - y_k = \max(j) + \min(v) = (y_o - 1) - 0 = y_o - 1 \end{cases}$
    $\Rightarrow y_o = y_i - y_k + 1$

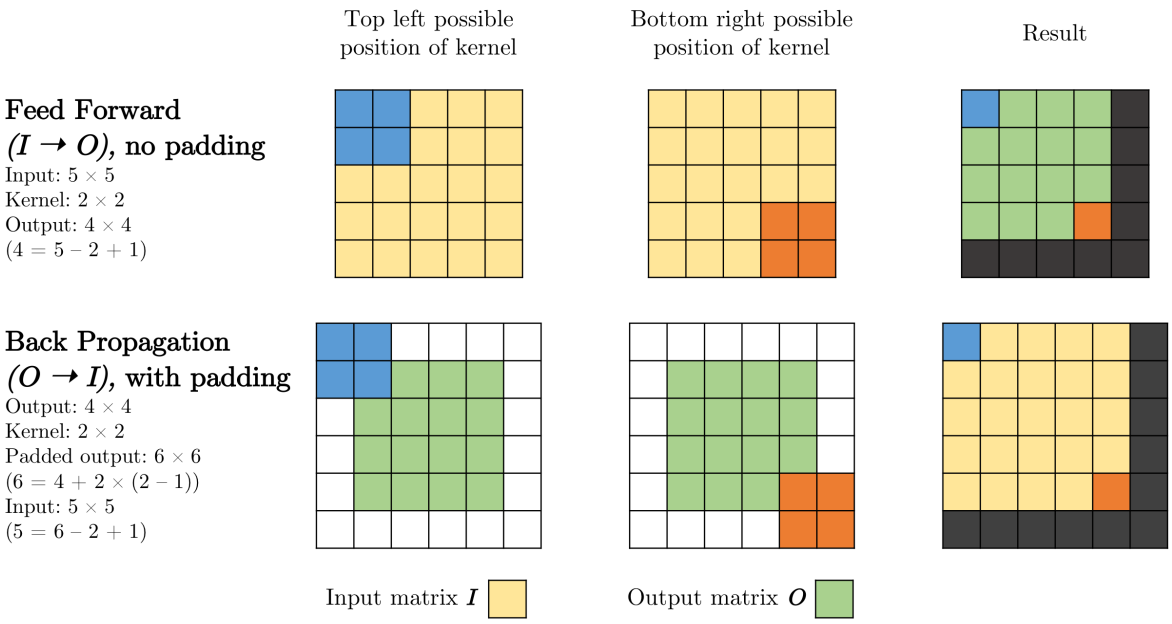
In both operations, the shape of $\boldsymbol{O}$ given $\boldsymbol{I}$ and $\boldsymbol{K}$ is $(x_o, y_o) = (x_i - x_k + 1, y_i - y_k + 1)$. However, it is evident that indexing in correlation is a lot cleaner than indexing in convolution. Correlation also follows how indexing works in most programming languages (indices of an array of size N should go from 0 to $N-1$). To meet the strict requirements of array indexing in coding, I replace convolution with correlation in the derivation below and in the implementation of a CNN. The kernels learned by the CNN using this implementation can be thought of as the 180^o-rotated versions of kernels that the CNN would have learned had I not changed convolution to correlation.

2.2.3 Padding

In the formulas above, for both correlation and convolution, the shape of the output is smaller than that of the input ($(x_o, y_o) = (x_i - x_k + 1, y_i - y_k + 1)$). If we think of correlation and convolution as sliding a kernel on top of the input, with these formulas, we need to ensure that the kernel never goes beyond the edges of the input. As a result, for any kernel of size larger than (1×1), the range of indices that can be the top-left of a kernel is smaller than the range of indices of the input matrix, as they have to account for width and height of the kernel. Therefore, the output matrix is smaller than the matrix.

In certain cases, it can be useful to pad the input matrix with all-zero edges, resulting in an output of a shape larger than that of the input. We will see later that **during feed forward, correlation without padding should be used**, while **during back propagation, convolution with padding should be used**. This makes sense in terms of:

- **Shapes**: if during feed forward, we reduce the shape of input $\boldsymbol{I}$ to the shape of output $\boldsymbol{O}$ by not using padding, then during back propagation (which moves in the exact opposite direction) we need to enlarge $\boldsymbol{O}$ to the shape of $\boldsymbol{I}$ by using padding.
- **Indexing**: See the figure below. In feed forward $\boldsymbol{I}(0,0)$ only contributes to $\boldsymbol{O}(0,0)$ and no other output cells. In back propagation, $\boldsymbol{I}(0,0)$ only receives contribution from $\boldsymbol{O}(0,0)$ and no other output cells. This is true for all other input cells on the edges: they only receive the propagation of loss gradient from output cells they contribute to.



Feed Forward with no padding and Back Propagation with padding preserve consistency in the shapes of input and output matrices

When padding is used:

- The shape of the padded input is:

$$(x_{pad,i}, y_{pad,i}) = (x_i + 2(x_k - 1), y_i + 2(y_k - 1))$$

This is because we add $(x_k - 1)$ rows to each of the top and bottom edges, and $(y_k - 1)$ to each of the left and right edges, just enough so that the kernel matrix can have at least one overlapping cell with the non-padded input matrix.

- The modified formula for convolution is:

$$\boldsymbol{O}(i,j) = \sum_{u=0}^{x_k} \sum_{v=0}^{y_k} \boldsymbol{I}_{pad}(i-u, j-v)\boldsymbol{K}(u,v) \quad \forall i \in [x_o], j \in [y_o]$$

$$\text{where } \boldsymbol{I}_{pad}(i-u, j-v) = \begin{cases} \boldsymbol{I}(i-u, j-v) \text{ if } 0 \le i-u \le x_i, 0 \le j-v \le y_i \\ 0 \text{ otherwise} \end{cases} \tag{3}$$

By adding padding, we can allow $(i-u, j-v)$ to be out of $\boldsymbol{I}$'s bounds: all out-of-bound indices are defined to be 0. Padding extends $\boldsymbol{I}$'s bounds from:

$$0 \le i - u \le x_i - 1$$
$$0 \le j - v \le y_i - 1$$

to $\boldsymbol{I_{pad}}$'s bounds:

$$-(x_k - 1) \le i - u \le x_i - 1 + (x_k - 1)$$
$$-(y_k - 1) \le j - v \le y_i - 1 + (y_k - 1)$$

According to equation (3), to make sure that the indices of $\boldsymbol{I}$ is not out of bound and that all indices in $\boldsymbol{I}$ is touched, we require that:

- $\forall i \in [x_o], u \in [x_k] : -(x_k - 1) \leq i + u \leq x_i - 1 + (x_k - 1)$
  $\Rightarrow \begin{cases} -(x_k - 1) = \min(i) - \max(u) = 0 - (x_k - 1), \text{ already satisfied} \\ x_i - 1 + (x_k - 1) = \max(i) - \min(u) = (x_o - 1) - 0 = x_o - 1 \end{cases}$
  $\Rightarrow x_o = x_i + x_k - 1$
- $\forall j \in [y_o], v \in [y_k] : -y_k + 1 \leq j + v \leq y_i - y_k$
  $\Rightarrow \begin{cases} -(y_k - 1) = \min(j) - \max(v) = 0 - (y_k + 1), \text{ already satisfied} \\ y_i - 1 + (y_k - 1) = \max(j) - \min(v) = (y_o - 1) - 0 = y_o - 1 \end{cases}$
  $\Rightarrow y_o = y_i + y_k - 1$

To recap, the shape of output matrix when padding is not used is:

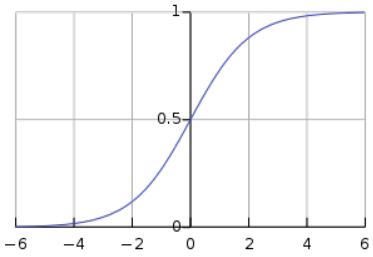
$$(x_o, y_o) = (x_i - x_k + 1, y_i - y_k + 1) \tag{4}$$

while the shape of output matrix when padding is used is:

$$(x_o, y_o) = (x_i + x_k - 1, y_i + y_k - 1) \tag{5}$$

2.3 Sigmoid function

Sigmoid function is a function with the characteristic S-shaped curve. It is widely used in neural network as an activation function, which models whether a neuron should fire (output a 1) or not fire (output a 0), given any input. It is usually applied right after the convolutional layer and right after the fully connected layer.



Sigmoid function

The function is defined as:

$$\sigma(x) = \frac{e^x}{e^x + 1} = \frac{1}{1 + e^{-x}}$$

Let's consider the derivative of this function. Define:

$$f(x) = \frac{1}{\sigma(x)} = 1 + e^{-x}$$

On one hand:

$$f'(x) = \frac{\partial}{\partial x}\left(\frac{1}{\sigma(x)}\right) = -\frac{\sigma'(x)}{\sigma^2(x)}$$

On the other hand:

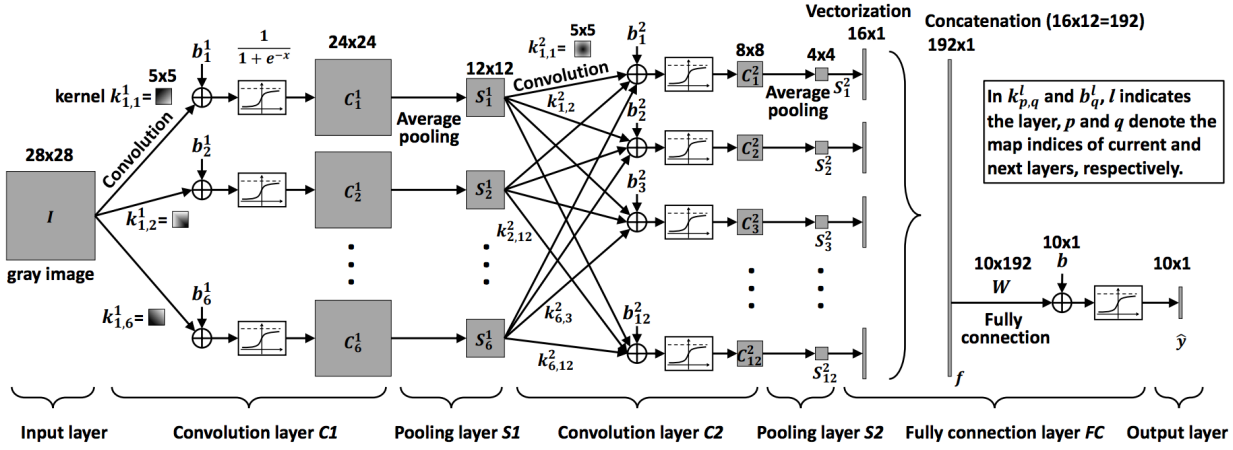
$$f'(x) = \frac{\partial}{\partial x}\left(1 + e^{-x}\right) = -e^{-x} = 1 - f(x) = 1 - \frac{1}{\sigma(x)} = \frac{\sigma(x) - 1}{\sigma(x)}$$

Equating the two equations, we have: $-\frac{\sigma'(x)}{\sigma^2(x)} = \frac{\sigma(x)-1}{\sigma(x)}$

$$\Leftrightarrow \sigma'(x) = (\sigma(x))(1 - \sigma(x)) \tag{6}$$

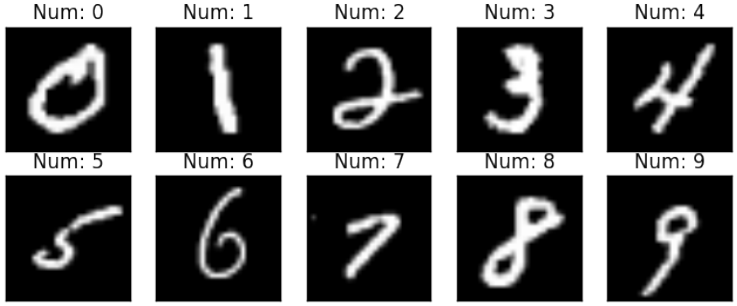
We will use this formula for the derivative of the sigmoid function in the derivation below.

3 Architecture of Convolutional Neural Networks



Architecture of the CNN used in this paper

The Convolutional Neural Network that we will use in this paper is described in the figure above. It has the same structure as the demo of Matlab DeepLearnToolbox. It takes in **28 × 28 inputs** and returns **10 × 1 outputs**. Thus, this CNN can be used to **classify handwritten digits** in the **MNIST** data set, which consists of **28 × 28 images** of digits belonging to **10 classes** (digits 0 to 9).



MNIST data set

4 Deriving Feed Forward and Back Propagation

4.1 Parameters Initialization

4.1.1 Convolutional Layer 1: $\boldsymbol{K}^1_{1,p}$ and b^1_p

$\forall p \in [6]$, $\boldsymbol{K}^1_{1,p}$ is a (5×5) matrix and b^1_p is a (1×1) scalar, initialized as:

$$\boldsymbol{K}^1_{1,p} \sim U\left(\pm\sqrt{\frac{6}{(1+6)\times 5^2}}\right) \tag{7}$$

$$b^1_p = 0 \tag{8}$$

4.1.2 Convolutional Layer 2: $\boldsymbol{K}^2_{p,q}$ and b^2_q

$\forall p \in [6], q \in [12]$, $\boldsymbol{K}^2_{p,q}$ is a (5×5) matrix and b^2_q is a (1×1) scalar, initialized as:

$$\boldsymbol{K}^2_{p,q} \sim U\left(\pm\sqrt{\frac{6}{(6+12)\times 5^2}}\right) \tag{9}$$

$$b^2_q = 0 \tag{10}$$

4.1.3 Fully Connected Layer: $\boldsymbol{W}$ and $\mathbf{b}$

$\boldsymbol{W}$ is a (10×192) matrix and $\mathbf{b}$ is a (10×1) scalar, initialized as:

$$\boldsymbol{W} \sim U\left(\pm\sqrt{\frac{6}{192+10}}\right) \tag{11}$$

$$\mathbf{b} = \mathbf{0} \tag{12}$$

4.2 Feed Forward

4.2.1 Convolutional Layer 1: $\boldsymbol{I} \to \boldsymbol{C}^1_\sigma$

$\boldsymbol{I}$: input, 28×28 matrix

$\boldsymbol{K}^1_{1,p}, p \in [6]$: kernel, 5×5 matrix

$\boldsymbol{B}^1_p, p \in [6]$: bias, 24×24 matrix whose all entries are of the same value, b^1_p

$\boldsymbol{C}^1_p, p \in [6]$: output before non-linearity, 24×24 matrix

$\boldsymbol{C}^1_{\sigma,p}, p \in [6]$: output after non-linearity, 24×24 matrix

$\forall p \in [6]$, we have:

$$\boldsymbol{C}_p^1 = \boldsymbol{I} \star \boldsymbol{K}_{1,p}^1 + \boldsymbol{B}_p^1 \tag{13}$$

$$\Leftrightarrow \boldsymbol{C}_p^1(i,j) = \sum_{u=0}^{3}\sum_{v=0}^{3}\Big(\boldsymbol{I}(i+u,j+v)\boldsymbol{K}_{1,p}^1(u,v)\Big) + b_p^1 \quad \forall i,j \in [24] \tag{14}$$

(based on correlation equation (1))

$$\boldsymbol{C}_{\sigma,p}^1 = \sigma(\boldsymbol{C}_p^1) \tag{15}$$

4.2.2 Pooling Layer 1: $\boldsymbol{C}_\sigma^1 \to \boldsymbol{S}^1$

$$\begin{aligned}\boldsymbol{C}_{\sigma,p}^1, p \in [6] &: \text{input}, 24 \times 24 \text{ matrix}\\ \boldsymbol{S}_p^1, p \in [6] &: \text{output}, 12 \times 12 \text{ matrix}\end{aligned}$$

$\forall p \in [6]$, we have:

$$\boldsymbol{S}_p^1(i,j) = \frac{1}{4}\sum_{u=0}^{1}\sum_{v=0}^{1}\boldsymbol{C}_{\sigma,p}^1(2i+u,2j+v) \quad \forall i,j \in [12] \tag{16}$$

4.2.3 Convolutional Layer 2: $\boldsymbol{S}^1 \to \boldsymbol{C}_\sigma^2$

$$\begin{aligned}\boldsymbol{S}_p^1, p \in [6] &: \text{input}, 12 \times 12 \text{ matrix}\\ \boldsymbol{K}_{p,q}^2, p \in [6], q \in [12] &: \text{kernel}, 5 \times 5 \text{ matrix}\\ \boldsymbol{B}_q^2, q \in [12] &: \text{bias}, 8 \times 8 \text{ matrix whose all entries are of the same value}, b_q^2\\ \boldsymbol{C}_q^2, q \in [12] &: \text{output before non-linearity}, 8 \times 8 \text{ matrix}\\ \boldsymbol{C}_{\sigma,q}^2, q \in [12] &: \text{output after non-linearity}, 8 \times 8 \text{ matrix}\end{aligned}$$

$\forall q \in [12]$, we have:

$$\boldsymbol{C}_q^2 = \sum_{p=0}^{5}\Big(\boldsymbol{S}_p^1 \star \boldsymbol{K}_{p,q}^2\Big) + \boldsymbol{B}_q^2 \tag{17}$$

$$\Leftrightarrow \boldsymbol{C}_q^2(i,j) = \sum_{p=0}^{5}\left(\sum_{u=0}^{3}\sum_{v=0}^{3}\boldsymbol{S}_p^1(i+u,j+v)\boldsymbol{K}_{p,q}^2(u,v)\right) + b_q^2 \quad \forall i,j \in [8] \tag{18}$$

(based on correlation equation (1))

$$\boldsymbol{C}_{\sigma,q}^2 = \sigma(\boldsymbol{C}_q^2) \tag{19}$$

4.2.4 Pooling Layer 2: $\boldsymbol{C}_{\sigma}^{2} \rightarrow \boldsymbol{S}^{2}$

$$\begin{aligned} \boldsymbol{C}_{\sigma,q}^{2}, q \in [12] : \text{input}, 8 \times 8 \text{ matrix} \\ \boldsymbol{S}_{q}^{2}, q \in [12] : \text{output}, 4 \times 4 \text{ matrix} \end{aligned}$$

$\forall q \in [12]$, we have:

$$\boldsymbol{S}_{q}^{2}(i,j) = \frac{1}{4} \sum_{u=0}^{1} \sum_{v=0}^{1} \boldsymbol{C}_{\sigma,q}^{2}(2i+u, 2j+v) \quad \forall i,j \in [4] \tag{20}$$

4.2.5 Vectorization and Concatenation: $\boldsymbol{S}^{2} \rightarrow \mathbf{v}$

$$\begin{aligned} \boldsymbol{S}_{q}^{2}, q \in [12] : \text{input}, 4 \times 4 \text{ matrix} \\ \mathbf{v} : \text{output}, 192 \times 1 \text{ vector} \end{aligned}$$

S_q^2 is an array of kernels. To visualize S_q^2, let's pretend that it has only 2 kernels, each of which is of size 2×2:

$$\begin{pmatrix} \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \\ \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix} \end{pmatrix}$$

After vectorizing S_q^2, we want to get:

$$\begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \end{pmatrix}$$

This is equivalent to "unrolling" each kernel in S_q^2 into a vector by going row by row, then combined the unrolled kernel into a long vector.
Now, let's define the concrete formula for this operation. We first vectorize each matrix S_q^2.

$\forall q \in [12]$, we have:

$$\mathbf{v}_q = \begin{pmatrix} | \\ \boldsymbol{S}_q^2(0,:)^\top \\ | \\ \dots \\ | \\ \boldsymbol{S}_q^2(3,:)^\top \\ | \end{pmatrix}$$

$$\Leftrightarrow \mathbf{v}_q(4i+j) = \boldsymbol{S}_q^2(i,j) \quad \forall i,j \in [4] \tag{21}$$

Now, we will concatenate the vectorized matrices into one long matrix.

$$\mathbf{v} = \begin{pmatrix} | \\ \mathbf{v}_1 \\ | \\ \dots \\ | \\ \mathbf{v}_{12} \\ | \end{pmatrix}$$

$$\Leftrightarrow \mathbf{v}(16q+4i+j) = \boldsymbol{S}_q^2(i,j) \quad \forall q \in [12], \forall i,j \in [4] \tag{22}$$

4.2.6 Fully Connected Layer: $\mathbf{v} \to \hat{\mathbf{y}}_\sigma$

$\mathbf{v}$: input, 192×1 vector
$\boldsymbol{W}$: weights, 10×192 matrix
$\mathbf{b}$: bias, 10×1 vector of independent values (unlike the biases above)
$\hat{\mathbf{y}}$: output before non-linearity, 10×1 vector
$\hat{\mathbf{y}}_\sigma$: output after non-linearity, 10×1 vector

$$\hat{\mathbf{y}} = \boldsymbol{W}\mathbf{v} + \mathbf{b} \tag{23}$$

$$\Leftrightarrow \hat{\mathbf{y}}(i) = \sum_{j=0}^{191} \Big(\boldsymbol{W}(i,j)\mathbf{v}(j) \Big) + \mathbf{b}(i) \tag{24}$$

$$\hat{\mathbf{y}}_\sigma = \sigma(\hat{y}) \tag{25}$$

4.3 Loss Evaluation

$\hat{\mathbf{y}}_\sigma$: predicted labels, 10×1 vector
$\mathbf{y}$: actual labels, 10×1 vector
L : loss, 1×1 scalar

$$L = \frac{1}{2} \left\| \hat{\mathbf{y}}_\sigma - \mathbf{y} \right\|^2 = \frac{1}{2} \sum_{i=0}^{9} (\hat{\mathbf{y}}_\sigma(i) - \mathbf{y}(i))^2 \tag{26}$$

4.4 Back Propagation

4.4.1 Fully Connected Layer: $\hat{\mathbf{y}}_\sigma \rightarrow \mathbf{v}$

- Calculating $\Delta\hat{\mathbf{y}}_\sigma$

$\Delta\hat{\mathbf{y}}_\sigma$: partial w.r.t predicted labels after non-linearity, 10×1 vector
$\hat{\mathbf{y}}_\sigma$: predicted labels after non-linearity, 10×1 vector
$\mathbf{y}$: actual labels, 10×1 vector

$$\begin{aligned}
\Delta\hat{\mathbf{y}}_\sigma(i) &= \frac{\partial L}{\partial \hat{\mathbf{y}}_\sigma(i)} \\
&= \frac{\partial}{\partial \hat{\mathbf{y}}_\sigma(i)} \left(\frac{1}{2} \sum_{\tilde{i}=0}^{9} (\hat{\mathbf{y}}_\sigma(\tilde{i}) - \mathbf{y}(\tilde{i}))^2 \right) \\
&\quad \text{(based on loss equation (26))} \\
&= \frac{1}{2} \sum_{\tilde{i}=0}^{9} \left(\frac{\partial}{\partial \hat{\mathbf{y}}_\sigma(i)} (\hat{\mathbf{y}}_\sigma(\tilde{i}) - \mathbf{y}(\tilde{i}))^2 \right)
\end{aligned}$$

Note that:

$$\frac{\partial}{\partial \hat{\mathbf{y}}_\sigma(i)} (\hat{\mathbf{y}}_\sigma(\tilde{i}) - \mathbf{y}(\tilde{i}))^2 = \begin{cases} 2\hat{\mathbf{y}}_\sigma(i) - \mathbf{y}(i) & \text{if } \tilde{i} = i \\ 0 \quad \text{if } \tilde{i} \neq i \end{cases}$$

Therefore:

$$\Delta\hat{\mathbf{y}}_\sigma(i) = \hat{\mathbf{y}}_\sigma(i) - \mathbf{y}(i) \tag{27}$$

- Calculating $\Delta\hat{\mathbf{y}}$

$\Delta\hat{\mathbf{y}}$: partial w.r.t. predicted labels before non-linearity, 10×1 vector
$\Delta\hat{\mathbf{y}}_\sigma$: partial w.r.t predicted labels after non-linearity, 10×1 vector
$\hat{\mathbf{y}}_\sigma$: predicted labels after non-linearity, 10×1 vector

$$\begin{aligned}
\Delta\hat{\mathbf{y}}(i) &= \frac{\partial L}{\partial \hat{\mathbf{y}}(i)} \\
&= \frac{\partial L}{\partial \hat{\mathbf{y}}_\sigma(i)} \cdot \frac{\partial \hat{\mathbf{y}}_\sigma(i)}{\partial \hat{\mathbf{y}}(i)} \\
&= \Delta\hat{\mathbf{y}}_\sigma(i) \cdot \hat{\mathbf{y}}_\sigma(i)(1 - \hat{\mathbf{y}}_\sigma(i)) \qquad (28) \\
&\quad \text{(based on sigmoid derivative equation (6))}
\end{aligned}$$

- Calculating $\Delta \boldsymbol{W}$

  $\Delta \boldsymbol{W}$: partial w.r.t weights, 10×192 matrix

  $\Delta \hat{\mathbf{y}}$: partial w.r.t. predicted labels before non-linearity, 10×1 vector

  $\mathbf{v}$: input of fully connected layer, 192×1 vector

$$
\begin{aligned}
\Delta \boldsymbol{W}(i,j) &= \frac{\partial L}{\partial \boldsymbol{W}(i,j)} \\
&= \frac{\partial L}{\partial \hat{\mathbf{y}}(i)} \cdot \frac{\partial \hat{\mathbf{y}}(i)}{\partial \boldsymbol{W}(i,j)} \\
&= \Delta \hat{\mathbf{y}}(i) \cdot \frac{\partial}{\partial \boldsymbol{W}(i,j)} \left(\sum_{\tilde{j}=0}^{191} \Big(\boldsymbol{W}(i,\tilde{j}) \mathbf{v}(\tilde{j}) \Big) + \mathbf{b}(i) \right) \\
&\quad \text{(based on } \hat{\mathbf{y}} \text{ equation (24))} \\
&= \Delta \hat{\mathbf{y}}(i) \cdot \sum_{\tilde{j}=0}^{191} \left(\frac{\partial}{\partial \boldsymbol{W}(i,j)} \Big(\boldsymbol{W}(i,\tilde{j}) \mathbf{v}(\tilde{j}) \Big) \right)
\end{aligned}
$$

  Note that:

$$
\frac{\partial}{\partial \boldsymbol{W}(i,j)} \Big(\boldsymbol{W}(i,\tilde{j}) \mathbf{v}(\tilde{j}) \Big) = \begin{cases} \mathbf{v}(j) & \text{if } \tilde{j} = j \\ 0 & \text{if } \tilde{j} \neq j \end{cases}
$$

  Therefore:

$$
\begin{aligned}
\Delta \boldsymbol{W}(i,j) &= \Delta \hat{\mathbf{y}}(i) \cdot \mathbf{v}(j) \\
\Rightarrow \Delta \boldsymbol{W} &= \Delta \hat{\mathbf{y}} \cdot \mathbf{v}^{\top}
\end{aligned} \tag{29}
$$

- Calculating $\Delta \mathbf{b}$

  $\Delta \mathbf{b}$: partial w.r.t. bias, 10×1 vector

  $\Delta \hat{\mathbf{y}}$: partial w.r.t. predicted labels before non-linearity, 10×1 vector

$$
\begin{aligned}
\Delta \mathbf{b}(i) &= \frac{\partial L}{\partial \mathbf{b}(i)} \\
&= \frac{\partial L}{\partial \hat{\mathbf{y}}(i)} \cdot \frac{\partial \hat{\mathbf{y}}(i)}{\partial \mathbf{b}(i)} \\
&= \Delta \hat{\mathbf{y}}(i) \cdot \frac{\partial}{\partial \mathbf{b}(i)} \left(\sum_{\tilde{j}=0}^{191} \Big(\boldsymbol{W}(i,\tilde{j}) \mathbf{v}(\tilde{j}) \Big) + \mathbf{b}(i) \right) \\
&\quad \text{(based on } \hat{\mathbf{y}} \text{ equation (24))} \\
&= \Delta \hat{\mathbf{y}}(i) \cdot 1 \\
\Rightarrow \Delta \mathbf{b} &= \Delta \hat{\mathbf{y}}
\end{aligned} \tag{30}
$$

4.4.2 Vectorization and Concatenation: $\mathbf{v} \rightarrow \boldsymbol{S}^2$

- Calculating $\Delta\mathbf{v}$

$\Delta\mathbf{v}$: partial w.r.t input of fully connected layer, 192×1 vector
$\boldsymbol{W}$: weights, 10×192 matrix
$\Delta\hat{\mathbf{y}}$: partial w.r.t. predicted labels before non-linearity, 10×1 vector

$$\begin{aligned}
\Delta\mathbf{v}(j) &= \frac{\partial L}{\partial \mathbf{v}(j)} \\
&= \sum_{i=0}^{9} \frac{\partial L}{\partial \hat{\mathbf{y}}(i)} \cdot \frac{\partial \hat{\mathbf{y}}(i)}{\partial \mathbf{v}(j)} \\
&= \sum_{i=0}^{9} \Delta\hat{\mathbf{y}}(i) \cdot \frac{\partial}{\partial \mathbf{v}(j)} \left(\sum_{\tilde{j}=0}^{191} \Big(\boldsymbol{W}(i,\tilde{j})\mathbf{v}(\tilde{j}) \Big) + \mathbf{b}(i) \right) \\
&\quad \text{(based on } \hat{\mathbf{y}} \text{ equation (24))} \\
&= \sum_{i=0}^{9} \Delta\hat{\mathbf{y}}(i) \cdot \sum_{\tilde{j}=0}^{191} \left(\frac{\partial}{\partial \mathbf{v}(j)} \Big(\boldsymbol{W}(i,\tilde{j})\mathbf{v}(\tilde{j}) \Big) \right)
\end{aligned}$$

Note that:

$$\frac{\partial}{\partial \mathbf{v}(j)} \Big(\boldsymbol{W}(i,\tilde{j})\mathbf{v}(\tilde{j}) \Big) = \begin{cases} \boldsymbol{W}(i,j) & \text{if } \tilde{j} = j \\ 0 & \text{if } \tilde{j} \neq j \end{cases}$$

Therefore:

$$\begin{aligned}
\Delta\mathbf{v}(j) &= \sum_{i=0}^{9} \Delta\hat{\mathbf{y}}(i) \cdot \boldsymbol{W}(i,j) \\
\Rightarrow \Delta\mathbf{v} &= \boldsymbol{W}^{\top} \cdot \Delta\hat{\mathbf{y}}
\end{aligned} \tag{31}$$

4.4.3 Pooling Layer 2: $\boldsymbol{S}^2 \rightarrow \boldsymbol{C}^2_{\sigma}$

- Calculating $\Delta\boldsymbol{S}^2_q$

$\Delta\boldsymbol{S}^2_q, q \in [12]$: parital w.r.t output of pooling layer $2, 4 \times 4$ matrix
$\Delta\mathbf{v}$: partial w.r.t input of fully connected layer, 192×1 vector

$$\Delta \mathbf{v}_q = \begin{pmatrix} | \\ \Delta \boldsymbol{S}_q^2(0,:)^\top \\ | \\ ... \\ | \\ \Delta \boldsymbol{S}_q^2(3,:)^\top \\ | \end{pmatrix}$$

$$\Leftrightarrow \Delta \boldsymbol{S}_q^2(i,j) = \Delta \mathbf{v}_q(4i+j) \quad \forall i,j \in [4] \tag{32}$$

(based on vectorization equation (21))

$$\Delta \mathbf{v} = \begin{pmatrix} | \\ \Delta \mathbf{v}_1 \\ | \\ ... \\ | \\ \Delta \mathbf{v}_{12} \\ | \end{pmatrix}$$

$$\Leftrightarrow \Delta \boldsymbol{S}_q^2(i,j) = \Delta \mathbf{v}(16q+4i+j) \quad \forall q \in [12], \forall i,j \in [4] \tag{33}$$

(based on concatenation equation (22))

4.4.4 Convolutional Layer 2: $\boldsymbol{C}_\sigma^2 \rightarrow \boldsymbol{S}^1$

- Calculating $\Delta \boldsymbol{C}_{\sigma,q}^2$

$$\Delta \boldsymbol{C}_{\sigma,q}^2, q \in [12] : \text{parital w.r.t output of convolutional layer 2 after non-linearity}, 8 \times 8 \text{ matrix}$$
$$\Delta \boldsymbol{S}_q^2, q \in [12] : \text{parital w.r.t output of pooling layer 2}, 4 \times 4 \text{ matrix}$$

Note that $\boldsymbol{C}_{\sigma,q}^2(i,j)$ only contributes to $\boldsymbol{S}_q^2(\lfloor i/2 \rfloor, \lfloor j/2 \rfloor)$, based on $\boldsymbol{S}_q^2$ equation (20). Therefore:

$$\begin{aligned}
\Delta \boldsymbol{C}_{\sigma,q}^2(i,j) &= \frac{\partial L}{\partial \boldsymbol{C}_{\sigma,q}^2(i,j)} \\
&= \frac{\partial L}{\partial \boldsymbol{S}_q^2(\lfloor i/2 \rfloor, \lfloor j/2 \rfloor))} \cdot \frac{\partial \boldsymbol{S}_q^2(\lfloor i/2 \rfloor, \lfloor j/2 \rfloor)}{\partial \boldsymbol{C}_{\sigma,q}^2(i,j)} \\
&= \Delta \boldsymbol{S}_q^2(\lfloor i/2 \rfloor, \lfloor j/2 \rfloor) \cdot \frac{\partial}{\partial \boldsymbol{C}_{\sigma,q}^2(i,j)} \left(\frac{1}{4} \sum_{u=0}^{1} \sum_{v=0}^{1} \boldsymbol{C}_{\sigma,q}^2(2\lfloor i/2 \rfloor + u, 2\lfloor j/2 \rfloor + v) \right) \\
&\quad \text{(based on } \boldsymbol{S}_q^2 \text{ equation (20))} \\
&= \Delta \boldsymbol{S}_q^2(\lfloor i/2 \rfloor, \lfloor j/2 \rfloor) \cdot \frac{1}{4} \\
&= \frac{1}{4} \Delta \boldsymbol{S}_q^2(\lfloor i/2 \rfloor, \lfloor j/2 \rfloor)
\end{aligned} \tag{34}$$

- Calculating $\Delta \boldsymbol{C}_q^2$

$$\begin{aligned}
\Delta \boldsymbol{C}_q^2, q \in [12] &: \text{parital w.r.t output of convolutional layer 2} \\
&\quad \text{before non-linearity}, 8 \times 8 \text{ matrix} \\
\Delta \boldsymbol{C}_{\sigma,q}^2, q \in [12] &: \text{parital w.r.t output of convolutional layer 2} \\
&\quad \text{after non-linearity}, 8 \times 8 \text{ matrix} \\
\boldsymbol{C}_{\sigma,q}^2, q \in [12] &: \text{output of convolutional layer 2} \\
&\quad \text{after non-linearity}, 8 \times 8 \text{ matrix}
\end{aligned}$$

$$\begin{aligned}
\Delta \boldsymbol{C}_q^2(i,j) &= \frac{\partial L}{\partial \boldsymbol{C}_q^2(i,j)} \\
&= \frac{\partial L}{\partial \boldsymbol{C}_{\sigma,q}^2(i,j)} \cdot \frac{\partial \boldsymbol{C}_{\sigma,q}^2(i,j)}{\partial \boldsymbol{C}_q^2(i,j)} \\
&= \Delta \boldsymbol{C}_{\sigma,q}^2(i,j) \cdot \boldsymbol{C}_{\sigma,q}^2(i,j) \left(1 - \boldsymbol{C}_{\sigma,q}^2(i,j)\right) \qquad (35) \\
&\quad \text{(based on sigmoid derivative equation (6))}
\end{aligned}$$

- Calculating $\Delta \boldsymbol{K}_{p,q}^2$

$$\begin{aligned}
\Delta \boldsymbol{K}_{p,q}^2, p \in [6], q \in [12] &: \text{partial w.r.t kernel of convolutional layer 2}, 5 \times 5 \text{ matrix} \\
\boldsymbol{S}_p^1, p \in [6] &: \text{output of pooling layer 1}, 12 \times 12 \text{ matrix} \\
\Delta \boldsymbol{C}_q^2, q \in [12] &: \text{parital w.r.t output of convolutional layer 2} \\
&\quad \text{before non-linearity}, 8 \times 8 \text{ matrix}
\end{aligned}$$

Note that every kernel $\boldsymbol{K}_{p,q}^2(u,v)$ contributes to every output cell $\boldsymbol{C}_q^2(i,j)$, based on $\boldsymbol{C}_q^2$ equation (18). Therefore:

$$\begin{aligned}
\Delta \boldsymbol{K}_{p,q}^2(u,v) &= \frac{\partial L}{\boldsymbol{K}_{p,q}^2(u,v)} \\
&= \sum_{i=0}^{7} \sum_{j=0}^{7} \frac{\partial L}{\partial \boldsymbol{C}_q^2(i,j)} \cdot \frac{\partial \boldsymbol{C}_q^2(i,j)}{\partial \boldsymbol{K}_{p,q}^2(u,v)} \\
&= \sum_{i=0}^{7} \sum_{j=0}^{7} \Delta \boldsymbol{C}_q^2(i,j) \\
&\quad \cdot \frac{\partial}{\partial \boldsymbol{K}_{p,q}^2(u,v)} \left(\sum_{p=0}^{5} \left(\sum_{\tilde{u}=0}^{3} \sum_{\tilde{v}=0}^{3} \boldsymbol{S}_p^1(i+\tilde{u}, j+\tilde{v}) \boldsymbol{K}_{p,q}^2(\tilde{u},\tilde{v}) \right) + b_q^2 \right) \\
&\quad \text{(based on } \boldsymbol{C}_q^2 \text{ equation (18))} \\
&= \sum_{i=0}^{7} \sum_{j=0}^{7} \Delta \boldsymbol{C}_q^2(i,j) \cdot \sum_{p=0}^{5} \sum_{\tilde{u}=0}^{3} \sum_{\tilde{v}=0}^{3} \left(\frac{\partial}{\partial \boldsymbol{K}_{p,q}^2(u,v)} \boldsymbol{S}_p^1(i+\tilde{u}, j+\tilde{v}) \boldsymbol{K}_{p,q}^2(\tilde{u},\tilde{v}) \right)
\end{aligned}$$

Note that:

$$\frac{\partial}{\partial \boldsymbol{K}^2_{p,q}(u,v)} \boldsymbol{S}^1_p(i+\tilde{u}, j+\tilde{v}) \boldsymbol{K}^2_{p,q}(\tilde{u},\tilde{v}) = \begin{cases} \boldsymbol{S}^1_p(i+u, j+v) & \text{if } \tilde{u}=u \text{ and } \tilde{v}=v \\ 0 & \text{otherwise} \end{cases}$$

Therefore:

$$\begin{aligned} \Delta \boldsymbol{K}^2_{p,q}(u,v) &= \sum_{i=0}^{7}\sum_{j=0}^{7} \Delta \boldsymbol{C}^2_q(i,j) \cdot \boldsymbol{S}^1_p(i+u, j+v) \\ &= \sum_{i=0}^{7}\sum_{j=0}^{7} \boldsymbol{S}^1_p(u+i, v+i) \cdot \Delta \boldsymbol{C}^2_q(i,j) \\ \Rightarrow \Delta \boldsymbol{K}^2_{p,q} &= \boldsymbol{S}^1_p \star \Delta \boldsymbol{C}^2_q \qquad (36) \\ &\quad \text{(based on correlation equation (1))} \end{aligned}$$

- Calculating Δb^2_q

$$\begin{aligned} \Delta b^2_q, q \in [12] &: \text{bias}, 1 \times 1 \text{ scalar} \\ \Delta \boldsymbol{C}^2_q, q \in [12] &: \text{parital w.r.t output of convolutional layer 2} \\ &\quad \text{before non-linearity}, 8 \times 8 \text{ matrix} \end{aligned}$$

Note that every bias b^2_q contributes to every output cell $\boldsymbol{C}^2_q(i,j)$, based on $\boldsymbol{C}^2_q$ equation (18). Therefore:

$$\begin{aligned} \Delta b^2_q &= \frac{\partial L}{\partial b^2_q} \\ &= \sum_{i=0}^{7}\sum_{j=0}^{7} \frac{\partial L}{\partial \boldsymbol{C}^2_q(i,j)} \cdot \frac{\partial \boldsymbol{C}^2_q(i,j)}{\partial b^2_q} \\ &= \sum_{i=0}^{7}\sum_{j=0}^{7} \Delta \boldsymbol{C}^2_q(i,j) \\ &\quad \cdot \frac{\partial}{\partial b^2_q} \left(\sum_{p=0}^{5} \left(\sum_{\tilde{u}=0}^{3}\sum_{\tilde{v}=0}^{3} \boldsymbol{S}^1_p(i+\tilde{u}, j+\tilde{v}) \boldsymbol{K}^2_{p,q}(\tilde{u},\tilde{v}) \right) + b^2_q \right) \\ &\quad \text{(based on } \boldsymbol{C}^2_q \text{ equation (18))} \\ &= \sum_{i=0}^{7}\sum_{j=0}^{7} \Delta \boldsymbol{C}^2_q(i,j) \cdot 1 \\ &= \sum_{i=0}^{7}\sum_{j=0}^{7} \Delta \boldsymbol{C}^2_q(i,j) \qquad (37) \end{aligned}$$

4.4.5 Pooling Layer 1: $\boldsymbol{S}^1 \rightarrow \boldsymbol{C}^1_\sigma$

- Calculating $\Delta \boldsymbol{S}^1_p$

$$\begin{aligned}
\Delta \boldsymbol{S}^1_p, p \in [6] &: \text{partial w.r.t. output of pooling layer } 1, 12 \times 12 \text{ matrix} \\
\Delta \boldsymbol{C}^2_q, q \in [12] &: \text{parital w.r.t output of convolutional layer } 2 \\
&\quad \text{before non-linearity}, 8 \times 8 \text{ matrix} \\
\boldsymbol{K}^2_{p,q}, p \in [6], q \in [12] &: \text{kernel of convolutional layer } 2, 5 \times 5 \text{ matrix}
\end{aligned}$$

Note that each input cell $\boldsymbol{S}^1_p(i,j)$ only contributes to output cells $\boldsymbol{C}^2_q(i-u, j-v)$ where $0 \leq u \leq 3$ and $0 \leq v \leq 3$, based on $\boldsymbol{C}^2_q$ equation (18). Therefore:

$$\begin{aligned}
\Delta \boldsymbol{S}^1_p(i,j) &= \frac{\partial L}{\partial \boldsymbol{S}^1_p(i,j)} \\
&= \sum_{q=0}^{11} \sum_{u=0}^{3} \sum_{v=0}^{3} \frac{\partial L}{\partial \boldsymbol{C}^2_q(i-u, j-v)} \cdot \frac{\partial \boldsymbol{C}^2_q(i-u, j-v)}{\partial \boldsymbol{S}^1_p(i,j)} \\
&= \sum_{q=0}^{11} \sum_{u=0}^{3} \sum_{v=0}^{3} \Delta \boldsymbol{C}^2_q(i-u, j-v) \\
&\quad \cdot \frac{\partial}{\partial \boldsymbol{S}^1_p(i,j)} \left(\sum_{p=0}^{5} \left(\sum_{\tilde{u}=0}^{3} \sum_{\tilde{v}=0}^{3} \boldsymbol{S}^1_p(i-u+\tilde{u}, j-v+\tilde{v}) \boldsymbol{K}^2_{p,q}(\tilde{u}, \tilde{v}) \right) + b^2_q \right) \\
&\quad \text{(based on } \boldsymbol{C}^2_q \text{ equation (18))} \\
&= \sum_{q=0}^{11} \sum_{u=0}^{3} \sum_{v=0}^{3} \Delta \boldsymbol{C}^2_q(i-u, j-v) \\
&\quad \cdot \sum_{p=0}^{5} \sum_{\tilde{u}=0}^{3} \sum_{\tilde{v}=0}^{3} \left(\frac{\partial}{\partial \boldsymbol{S}^1_p(i,j)} \boldsymbol{S}^1_p(i-u+\tilde{u}, j-v+\tilde{v}) \boldsymbol{K}^2_{p,q}(\tilde{u}, \tilde{v}) \right)
\end{aligned}$$

Note that:

$$\begin{aligned}
&\frac{\partial}{\partial \boldsymbol{S}^1_p(i,j)} \boldsymbol{S}^1_p(i-u+\tilde{u}, j-v+\tilde{v}) \boldsymbol{K}^2_{p,q}(\tilde{u}, \tilde{v}) \\
&= \begin{cases} \boldsymbol{K}^2_{p,q}(u,v) & \text{if } (i-u+\tilde{u}) = i \text{ and } (j-v+\tilde{v}) = v \Leftrightarrow \tilde{u} = u \text{ and } \tilde{v} = v \\ 0 & \text{otherwise} \end{cases}
\end{aligned}$$

Therefore:

$$\begin{aligned}
\Delta \boldsymbol{S}^1_p(i,j) &= \sum_{q=0}^{11} \sum_{u=0}^{3} \sum_{v=0}^{3} \Delta \boldsymbol{C}^2_q(i-u, j-v) \cdot \boldsymbol{K}^2_{p,q}(u,v) \\
\Rightarrow \Delta \boldsymbol{S}^1_p &= \sum_{q=0}^{11} \Delta \boldsymbol{C}^2_q * \boldsymbol{K}^2_{p,q} \qquad (38) \\
&\quad \text{(based on convolution equation (2))}
\end{aligned}$$

4.4.6 Convolutional Layer 1: $\boldsymbol{C}^1_\sigma \to \boldsymbol{I}$

- Calculating $\Delta \boldsymbol{C}^1_{\sigma,p}$

$$\begin{aligned}
\Delta \boldsymbol{C}^1_p, p \in [6] &: \text{parital w.r.t output of convolutional layer 1} \\
&\quad \text{before non-linearity}, 24 \times 24 \text{ matrix} \\
\Delta \boldsymbol{C}^1_{\sigma,p}, p \in [6] &: \text{parital w.r.t output of convolutional layer 1} \\
&\quad \text{after non-linearity}, 24 \times 24 \text{ matrix} \\
\boldsymbol{C}^1_{\sigma,p}, p \in [6] &: \text{output of convolutional layer 1} \\
&\quad \text{after non-linearity}, 24 \times 24 \text{ matrix}
\end{aligned}$$

Note that $\boldsymbol{C}^1_{\sigma,p}(i,j)$ only contributes to $\boldsymbol{S}^1_p(\lfloor i/2 \rfloor, \lfloor j/2 \rfloor)$, based on $\boldsymbol{S}^1_p$ equation (16). Therefore:

$$\begin{aligned}
\Delta \boldsymbol{C}^1_{\sigma,p}(i,j) &= \frac{\partial L}{\partial \boldsymbol{C}^1_{\sigma,p}(i,j)} \\
&= \frac{\partial L}{\partial \boldsymbol{S}^1_p(\lfloor i/2 \rfloor, \lfloor j/2 \rfloor))} \cdot \frac{\partial \boldsymbol{S}^1_p(\lfloor i/2 \rfloor, \lfloor j/2 \rfloor)}{\partial \boldsymbol{C}^1_{\sigma,p}(i,j)} \\
&= \Delta \boldsymbol{S}^1_p(\lfloor i/2 \rfloor, \lfloor j/2 \rfloor) \cdot \frac{\partial}{\partial \boldsymbol{C}^1_{\sigma,p}(i,j)} \left(\frac{1}{4} \sum_{u=0}^{1} \sum_{v=0}^{1} \boldsymbol{C}^1_{\sigma,p}\left(2\lfloor i/2 \rfloor + u, 2\lfloor j/2 \rfloor + v\right) \right) \\
&\quad \text{(based on } \boldsymbol{S}^1_p \text{ equation (16))} \\
&= \Delta \boldsymbol{S}^1_p(\lfloor i/2 \rfloor, \lfloor j/2 \rfloor) \cdot \frac{1}{4} \\
&= \frac{1}{4} \Delta \boldsymbol{S}^1_p(\lfloor i/2 \rfloor, \lfloor j/2 \rfloor)
\end{aligned} \tag{39}$$

- Calculating $\Delta \boldsymbol{C}^1_p$

$$\begin{aligned}
\Delta \boldsymbol{C}^1_p, p \in [6] &: \text{parital w.r.t output of convolutional layer 1} \\
&\quad \text{before non-linearity}, 24 \times 24 \text{ matrix} \\
\Delta \boldsymbol{C}^1_{\sigma,p}, p \in [6] &: \text{parital w.r.t output of convolutional layer 1} \\
&\quad \text{after non-linearity}, 24 \times 24 \text{ matrix} \\
\boldsymbol{C}^1_{\sigma,p}, p \in [6] &: \text{output of convolutional layer 1} \\
&\quad \text{after non-linearity}, 24 \times 24 \text{ matrix}
\end{aligned}$$

$$\begin{aligned}
\Delta \boldsymbol{C}^1_p(i,j) &= \frac{\partial L}{\partial \boldsymbol{C}^1_p(i,j)} \\
&= \frac{\partial L}{\partial \boldsymbol{C}^1_{\sigma,p}(i,j)} \cdot \frac{\partial \boldsymbol{C}^1_{\sigma,p}(i,j)}{\partial \boldsymbol{C}^1_p(i,j)} \\
&= \Delta \boldsymbol{C}^1_{\sigma,p}(i,j) \cdot \boldsymbol{C}^1_{\sigma,p}(i,j) \left(1 - \boldsymbol{C}^1_{\sigma,p}(i,j)\right) \\
&\quad \text{(based on sigmoid derivative equation (6))}
\end{aligned} \tag{40}$$

- Calculating $\Delta \boldsymbol{K}^1_{1,p}$

$$\begin{aligned}
\Delta \boldsymbol{K}^1_{1,p}, p \in [6] &: \text{partial w.r.t kernel of convolutional layer } 1, 5 \times 5 \text{ matrix} \\
\boldsymbol{I} &: \text{input of the CNN}, 28 \times 28 \text{ matrix} \\
\Delta \boldsymbol{C}^1_p, p \in [6] &: \text{parital w.r.t output of convolutional layer 1} \\
&\quad \text{before non-linearity}, 24 \times 24 \text{ matrix}
\end{aligned}$$

Note that every kernel $\boldsymbol{K}^1_{1,p}(u,v)$ contributes to every output cell $\boldsymbol{C}^1_p(i,j)$, based on $\boldsymbol{C}^1_p$ equation (14). Therefore:

$$\begin{aligned}
\Delta \boldsymbol{K}^1_{1,q}(u,v) &= \frac{\partial L}{\boldsymbol{K}^1_{1,q}(u,v)} \\
&= \sum_{i=0}^{23}\sum_{j=0}^{23} \frac{\partial L}{\partial \boldsymbol{C}^1_p(i,j)} \cdot \frac{\partial \boldsymbol{C}^1_p(i,j)}{\partial \boldsymbol{K}^1_{1,q}(u,v)} \\
&= \sum_{i=0}^{23}\sum_{j=0}^{23} \Delta \boldsymbol{C}^1_p(i,j) \\
&\quad \cdot \frac{\partial}{\partial \boldsymbol{K}^1_{1,q}(u,v)} \left(\sum_{\tilde{u}=0}^{4}\sum_{\tilde{v}=0}^{4} \boldsymbol{I}(i+\tilde{u}, j+\tilde{v}) \boldsymbol{K}^1_{1,q}(\tilde{u},\tilde{v}) + b^1_p \right) \\
&\quad \text{(based on } \boldsymbol{C}^1_p \text{ equation (14))} \\
&= \sum_{i=0}^{23}\sum_{j=0}^{23} \Delta \boldsymbol{C}^1_p(i,j) \cdot \sum_{\tilde{u}=0}^{4}\sum_{\tilde{v}=0}^{4} \left(\frac{\partial}{\partial \boldsymbol{K}^1_{1,q}(u,v)} \boldsymbol{I}(i+\tilde{u}, j+\tilde{v}) \boldsymbol{K}^1_{1,q}(\tilde{u},\tilde{v}) \right)
\end{aligned}$$

Note that:

$$\frac{\partial}{\partial \boldsymbol{K}^1_{1,q}(u,v)} \boldsymbol{I}(i+\tilde{u}, j+\tilde{v}) \boldsymbol{K}^1_{1,q}(\tilde{u},\tilde{v}) = \begin{cases} \boldsymbol{I}(i+u, j+v) & \text{if } \tilde{u}=u \text{ and } \tilde{v}=v \\ 0 & \text{otherwise} \end{cases}$$

Therefore:

$$\begin{aligned}
&= \sum_{i=0}^{23}\sum_{j=0}^{23} \Delta \boldsymbol{C}^1_p(i,j) \cdot \boldsymbol{I}(i+u, j+v) \\
&= \sum_{i=0}^{23}\sum_{j=0}^{23} \boldsymbol{I}(u+i, v+i) \cdot \Delta \boldsymbol{C}^1_p(i,j) \\
\Rightarrow \Delta \boldsymbol{K}^1_{1,q} &= \boldsymbol{I} \star \Delta \boldsymbol{C}^1_p \qquad (41) \\
&\quad \text{(based on correlation equation (1))}
\end{aligned}$$

- Calculating Δb_p^1

$$\begin{aligned}\Delta b_p^1, p \in [6] &: \text{bias}, 1 \times 1 \text{ scalar} \\ \Delta \boldsymbol{C}_p^1, p \in [6] &: \text{parital w.r.t output of convolutional layer 1} \\ &\quad \text{before non-linearity}, 24 \times 24 \text{ matrix}\end{aligned}$$

Note that every bias b_p^1 contributes to every output cell $\boldsymbol{C}_p^1(i,j)$, based on $\boldsymbol{C}_p^1$ equation (14). Therefore:

$$\begin{aligned}\Delta b_p^1 &= \frac{\partial L}{\partial b_p^1} \\ &= \sum_{i=0}^{23}\sum_{j=0}^{23} \frac{\partial L}{\partial \boldsymbol{C}_p^1(i,j)} \cdot \frac{\partial \boldsymbol{C}_p^1(i,j)}{\partial b_p^1} \\ &= \sum_{i=0}^{23}\sum_{j=0}^{23} \Delta \boldsymbol{C}_p^1(i,j) \\ &\quad \cdot \frac{\partial}{\partial b_p^1}\left(\sum_{\tilde{u}=0}^{4}\sum_{\tilde{v}=0}^{4} \boldsymbol{I}(i+\tilde{u}, j+\tilde{v}) \boldsymbol{K}_{1,q}^1(\tilde{u},\tilde{v}) + b_p^1\right) \\ &\quad (\text{based on } \boldsymbol{C}_p^1 \text{ equation (14)}) \\ &= \sum_{i=0}^{23}\sum_{j=0}^{23} \Delta \boldsymbol{C}_p^1(i,j) \cdot 1 \\ &= \sum_{i=0}^{23}\sum_{j=0}^{23} \Delta \boldsymbol{C}_p^1(i,j) \end{aligned} \tag{42}$$

4.5 Parameters Update

Once we have the partial derivative with respect to each parameter being learned, we can perform **gradient descent** by adjusting these parameters in the negative gradient direction.

$$\begin{aligned}\boldsymbol{K}_{1,p}^1 &\leftarrow \boldsymbol{K}_{1,p}^1 - \alpha \cdot \Delta \boldsymbol{K}_{1,p}^1 \\ b_p^1 &\leftarrow b_p^1 - \alpha \cdot \Delta b_p^1 \\ \boldsymbol{K}_{p,q}^2 &\leftarrow \boldsymbol{K}_{p,q}^2 - \alpha \cdot \Delta \boldsymbol{K}_{p,q}^2 \\ b_q^2 &\leftarrow b_q^2 - \alpha \cdot \Delta b_q^2 \\ \boldsymbol{W} &\leftarrow \boldsymbol{W} - \alpha \cdot \Delta \boldsymbol{W} \\ \mathbf{b} &\leftarrow \mathbf{b} - \alpha \cdot \Delta \mathbf{b}\end{aligned}$$

5 Implementation

The implementation in Python of this paper is available on Github at:

https://github.com/nganvu/VanillaCNN

This implementation follows the derivation above as much as possible and therefore it is not optimized. It takes about 14 hours to go through all $60,000$ training data points in the MNIST data set (i.e. 14 hours / epoch). After letting my implementation run for one epoch, I got an accuracy of 0.10 (not better than random guess), even though the loss is consistently decreasing (especially at the beginning, before starting to slow down), which means the parameters are indeed going in the negative gradient direction of loss.

To investigate why this is the case, I used Keras, an API optimized for building and training deep learning models, to create a CNN that has the exact same architecture but runs a lot faster. After 1 epoch, the accuracy achieved by the Keras CNN is also just 0.10. This means that my implementation does not perform worse than the implementation of Keras, at least for one epoch. After 10 epochs, however, the accuracy is around 0.88 and still increasing. This suggests that I can let my implementation run for several days to see if the accuracy improves over a long period of time.

Below are several ideas I have to continue improving this project:

- Parallelize the computation of individual training data points, as they are completely independent of one another.
- Replace the sigmoid function in the Fully Connected Layer with the softmax function, which is often used for categorical predictions like with MNIST.
- Implement better variations of gradient descent (e.g. incorporating momentum, adjusting learning rate over time, etc.)
- Write unit tests to ensure each mathematical operation was implemented correctly.

References

Zhang, Zhifei. "Derivation of Backpropagation in Convolutional Neural Network (CNN)". *University of Tennessee, Knoxvill, TN* (October 18, 2016).

While inspired by the paper above, this paper:

- Provides more mathematical background for CNN.
- Explains every step of the derivation, especially in the Back Propagation section.
- Uses a very rigorous indexing scheme (and as a side effect, arrives at different formulas for some variables), allowing all formulas to be translated directly to code.
- Is accompanied by executable code that closely follows the derivation.